D

Towards Detecting Intrusions in a Networked Environment

L. Todd Heberlein, Biswanath Mukherjee
Karl Levitt, Gihan Dias

Computer Security Laboratory
Division of Computer Science
University of California
Davis, California 95616
(916) 752-2149
heberlei@iris.eecs.ucdavis.edu

Douglass Mansur

Lawrence Livermore National Laboratory
Livermore, Ca. 94550

Abstract

To date, current access control mechanisms have been shown to be insufficient for preventing intrusive activity in computer systems. The frequent media reports, and now our own research, have shown the widespread proliferation of intrusive behavior on the world's computer systems. With the recognition of the failure of access control mechanisms, a number of institutions have begun to research methods to detect the intrusive activity. The majority of this research has focused on analyzing audit trails generated by operating systems. We, on the other hand, have chosen to analyze the traffic on computer networks. The result of our effort is a suite of tools, collectively called the Network Security Monitor (NSM). This paper presents the methods used to represent and analyze traffic on the network for detecting intrusive activity. We also present two tools designed to further analyze network connections deemed intrusive. Finally, we present the successful and surprising results from the NSM.

1 INTRODUCTION

To date, authentication and access control mechanisms have been shown to be insufficient for preventing intrusive behavior in computer systems [2,13]. Almost weekly reports about outsiders breaking into computers or about employees misusing computer systems appear in the media. With the proliferation of both computer

networks as well as workstations and personal computers connected to them, more and more people will continue to have electronic paths to a larger number of computers. Furthermore, these computer systems are often administered by people with little or no formal training in system administration or computer security. The result is that more people have access to a larger number of possibly less secure computers. There is little doubt that intrusions in computer systems will continue.

With the recognition that current access control mechanisms and administrative policies have not provided adequate security, a number of projects have been started to develop systems that can identify intrusive behavior in a computer system [2,5,7,9,11,12,13,14]. These systems, called intrusion detection systems (IDS), analyze records of computer activity to identify potential intrusions. The records of computer activity are generally audit trails generated by host operating systems. Unfortunately, by using an operating system generated audit trail, an IDS can be restricted to a particular operating system. Furthermore, audit trails have themselves been the target of subversion by intruders, thereby rendering the analysis of the audit trails suspect. Finally, due to the processing and storage costs associated with the collection of audit trails, many administrators choose not to collect audit information.

Lawrence Livermore National Laboratory and the University of California at Davis recognized the growth of computer networks, the heterogeneous nature of the computer environment (everything from Crays to personal computers use the same networks), the need for the immediate deployment of intrusions detection systems, and the potential drawbacks of audit trail based IDS. Consequently, we began to study the potential of analyzing network information. By developing an IDS which monitors data transmitted on a broadcast Local Area Network, we could analyze intrusions on many different types of hosts. Furthermore, the network based IDS would not be affected by either the quality of the administration given to the hosts on the network or the security of the audit trails, and we would not necessarily burden the hosts being monitored. The result of this on-going research is a system called the Network Security Monitor (NSM)[7]. The NSM is actually a suite of tools for detecting and analyzing computer abuse. The methods by which the NSM detects intrusions, the tools to analyze the intrusions, and the promising early results from the NSM are the focus of this paper.

In section 2, we provide a description of the environment for which the NSM is developed, and we provide the requirements and constraints for the development of the NSM. In section 3, we provide the theoretical framework underlying the detection of intrusive behavior in a network. In section 4 we relate the theory introduced in section 3 to the particular problem of detecting intrusions in

our networked environment. In section 5, we present two tools to perform further analysis on a particular network connection. In section 6, we present results from using the NSM. And in section 7, we discuss future research related to our work with the NSM.

2 ENVIRONMENT AND REQUIREMENTS

In this section we present the environment for which the NSM is targeted and the requirements and constraints placed upon the development of the NSM.

The target environment, which needs to be protected from intrusive activity, consists of a number of host computers (including devices such as file servers, name servers, printers, etc) and a LAN through which the hosts are interconnected. The hosts consist of a variety of hardware platforms and operating systems. The LAN is assumed to employ a broadcast medium (eg., Ethernet), and all packets transmitted over the LAN are potentially available to any device connected to the network. The LAN is also assumed to be physically secure, in the sense that an intruder will not be able to directly access the network hardware such as the connecting medium (cable) and the network interface at each host. The LAN is connected to the outside world via one or more gateways.

The hosts on the LAN may connect, or attempt to connect, to any other host at any time by any network service. The hosts can generate as many connections as they wish, and connections can last for any amount of time. Hosts may be added or removed from the network at any time.

The NSM designed to monitor this complex and dynamic environment must be built from off-the-shelf hardware. Furthermore, the software must be portable to a variety of platforms. Finally, the NSM must be capable of detecting currently known types of intrusive behavior, and it must provide mechanisms for the detection of previously unknown intrusive behavior.

3 DETECTING BEHAVIOR IN A SYSTEM

This section provides the theoretical framework for the NSM's intrusion detection mechanism. The problem of detecting intrusions in the interconnected computing environment is abstracted out to detecting any behavior in a complex system. A methodology for solving this abstract problem is presented.

The problem we must solve is the identification of intrusive behavior in the interconnected computing environment (ICE). Since the ICE consists of network packets, network connections, hosts, bridges, and other components working in some orderly fashion, the ICE can be considered a "system," where a system is defined to be "a group of things or parts connected in some way as to form a whole." [6] The ICE system is both complex, in that it is composed of sub-parts which are in turn systems, and it is

dynamic, in that the composition of the system changes over time. Our problem can therefore be generalized to finding a particular behavior in a complex and dynamic system. If a solution can be found to the generalized problem, then a solution to our specific problem is reduced to defining the the ICE as a system.

By providing a solution to the general problem of finding behaviors in systems, we provide a methodology which is useful beyond the scope of detecting intrusions in a networked environment. Furthermore, a definition, in the format given below, of any complex and dynamic system provides the specification needed to implement behavior detection software for that system. Finally, the specification of the system provided by the definition may be strong enough to provide for automatic code generation for the majority of the behavior detection software.

Section 3.1 provides a brief description of attribute grammars, the ancestors of our system description language, and section 3.2 continues the discussion with the introduction to the system description language.

3.1 Attribute Grammars

Attribute grammars describe both the strings accepted by a language and a method to determine the "meaning" of those strings. An attribute grammar consist of a context free grammar, a set of attributes for each symbol in the grammar, and a set of functions defined within the scope of a production rule in the grammar to determine the values for the attributes of each symbol in that production. [1] The following example of an attribute grammar for the definition and interpretation of binary numbers* will be used to clarify the relationships between three components of an attribute grammar.

The context free grammar for our language of binary numbers is defined by G = (V,N,P,S) where V is the set of symbols, N is the set of nonterminal symbols, P is the set of production rules, and S, an element of N, is the start symbol. The set of terminal symbols, a subset of V, is {1,0,.}. These are the ASCII characters one, zero, and period. The set of nonterminal symbols, N, is {B,L,N}. They represent the abstract objects bit, list of bits, and number. The start symbol for our attribute grammar for binary numbers is N, the abstract number. The set of production rules relating these symbols and providing the definition of acceptable strings is given in figure 1.

By this context free grammar, we can see that the string 11.01 is an acceptable binary number. The parse tree for this string is given in figure 2.

The context free grammar can be used to build a parse tree of a string and determine whether the string is valid in the language; however, the context free grammar can

* This example is taken from [8]

not be used to find the meaning of the string. The addition of attributes and attribute functions are necessary to determine the meaning of the string.

The set of attributes, A, for each for each nonterminal are given as follows: A(B) = {*v*}, A(L) = {*v*,*l*}, and A(N) = {*v*}. The attribute *v* is the value of a symbol, and the attribute *l* is the length of a symbol.

The set of functions defined with in the scope of each production rule is given in figure 3.

By using the attributes for each symbol and the attribute functions, we can now assign meaning to each symbol in the parse tree (see figure 4). For our language of binary numbers, the most important meaning is that of the start symbol N. Our string 11.01 now has the meaning of 3.25.

3.2 System description language

This section introduces the system description language, an extension of attribute grammars. The system description language provides the structural definition of a system, the "meaning" of the system and system components, and a means to determine if a behavior is present in any part of the system.

A system description language consists of a structural grammar, a set of attributes for each object, or symbol, in the structural grammar, a set of functions defined within the scope of a production rule of the structural grammar to determine the attribute values for each object in that production, and behavior detection functions defined for each object in the structural grammar. The set of attributes for each object and the set of attribute functions are quite similar to those of attribute grammars, so they will not be discussed further. On the other hand, the structural grammar is different than that of context-free grammars, and behavior detection functions do not exist in attribute grammars; therefor, these elements of the system description language will be discussed in further detail.

3.2.1 Structural grammar

The structural grammar, similar to a context-free grammar, provides the information about the structure of the system. However, unlike a context-free grammar, the structural grammar requires attribute information to determine the parse tree for the system. The structural information of a system is defined by the structural grammar $G = (O,C,S,P)$, where O is the set of all objects; $C \subset O$ is the set of complex objects; $S \in C$ is the system level object; and P is the set of production rules giving the relationships between the elements of O. Each of these sets are described in more detail below.

O is the set of objects which compose the system. The set can be partitioned into two subsets: a set of basic

objects and a set of complex objects. Basic objects are atomic elements, and, therefore, they cannot be decomposed. Basic objects are the only objects of the system which can be directly observed. They are similar to terminal symbols in traditional programming languages.

C, a subset of O, is the set of complex objects in the system. Complex objects are composed of other complex objects and/or basic objects. Complex objects are abstract objects and are not observed directly; they must be created by assembling basic objects into the necessary format. Complex objects are similar to non-terminal symbols in traditional programming languages.

S, an element of C, is the system level object. S is the complex objects which represents the entire system. S cannot be an element of any other complex objects, so it cannot occur on the right hand side of any production. S, the system level object, is similar to the start symbol in traditional programming languages.

P, the set of production rules, provides the acceptable relationships between objects in the system. Production rules for a system, although similar to production rules for context-free grammars, require knowledge about some of the attributes for each of these objects. For example, the production rule

OBJECT_1 -> {OBJECTS_2}

states that an element of type OBJECT_1 is a complex object composed of zero or more complex objects of type OBJECT_2. Unlike context-free grammars, the set of elements of type OBJECT_2 which compose an element of type OBJECT_1 is based not on the order of appearance of the other elements, but on the certain attributes of the elements of OBJECT_1 and OBJECT2. To include this information, our production rule must be modified to include this information, so our production rule becomes:

```
OBJECT_1 -> {OBJECTS_2}
    where for each o2 ∈ {OBJECTS_2}
      OBJECT_1.atribute_m = o2.attribute_n
      OBJECT_1.attrubite_m+1 = o2.attribute_n+1
      ...
      OBJECT_1.attribute_m+i = o2.attribute_n+i
```

The production rule now states that an element of type OBJECT_1 is composed of a set of elements of type OBJECT_2 such that all elements of type OBJECT_2 have particular attributes which equal particular attributes of the element of type OBJECT_1. In general, elements of {OBJECTS_2} will have attributes which must satisfy a particular set of conditions. The set of conditions may or may not be based on attributes in OBJECT_1.

Another manner in which the structural grammar differs from context free grammars is that structural

grammars include the notion of time, and this notion of time introduces dynamics to a system parse tree. An element in a system can be considered active or inactive depending on the time which has passed since the element's attributes have changed. An inactive element may be pruned from the system parse tree. The notions of active and inactive can be used in the calculation of an object's attributes. This will be seen later.

3.2.2 Behavior detection functions

Once the structural grammar, attributes, and attribute functions have been defined, a second set of functions, behavior detection functions, must be defined for each object in the structural grammar. Behavior detection functions determine whether an object is associated with the particular behavior of interest. Because a behavior may manifest itself differently or more clearly in different object types, each object in a system parse tree must be examined for the behavior by particular behavior detection functions designed for that object type. For each type of object, there will be two behavior detection functions: the isolated behavior detection function, and the integrated behavior detection functions. These two function types are discussed below.

An isolated behavior detection function for an object uses the attributes of that object to calculate the probability that the object is associated with the behavior of interest. In short, an isolated behavior detection function is a classifier. With some preprocessing to transform the attribute types, a large number of classifiers can be used.

Unfortunately, classifiers generally have to be trained with sample data, and the behavior of interest is often quite rare. There are at least two possible solutions to the problem of lack of sample data: expert systems and single behavior classifiers. An expert system, designed by people knowledgeable about the problem domain, can use heuristics to determine how close an object is to the behavior of interest. A single behavior classifier is built around the assumption that a rare behavior will be significantly different than normal behavior. If this is true, a single classifier can profile normal behavior, and then it could report any behavior which does not strongly resemble normal behavior. Work on such single behavior classifiers have been done by SRI for IDES and Los Alamos National Laboratories for Wisdom and Sense. For our particular problem environment, we combined the efforts of both an expert system and a single behavior classifier.

An integrated behavior detection function for an object modifies the result of the isolated behavior detection function by including the analysis of the isolated behavior detection functions for sub-componenents and super-components. Sub-components and super components are define by the following rule:

If object A is a component of object B, then A is a sub-component of B, and B is a super-component of A.

The modification by an integrated behavior detection function allows the inclusion of both the results of aggregated analysis (those from super-components) and the results of more detailed levels of analysis (those from sub-components). The integrated behavior detection function can be implemented by a weighted average function (see figure 5).

The relationship between an object's attributes, the isolated behavior detection functions, and integrated behavior detection functions can be seen in figure 6. In this example, we are interested in analyzing the object B_1 for a particular behavior. The object B_1 is composed of objects C_1 and C_2, and it is part of the object A_1. Result B_{1V} is the analysis of object B_1 in isolation, and result $B_{1V'}$ is the result after combining the result of B_{1V} with the results from objects C_1, C_2, and A_1.

What has been presented up to this point is a methodology to detect behaviors in complex and dynamic systems. The system description language provides both a method to model a system and a specification for software to detect a particular behavior in that system. The next step is to apply the system description language to our interconnected computing environment.

4 DETECTING INTRUSIONS IN AN ICE

To detect intrusions in an interconnected computing environment, we use the system description language to model the ICE, and we choose intrusiveness to be the behavior of interest. The model of the ICE will be called the interconnected computing environment model, or ICEM. Section 4.1 provides a high level, English description of the ICEM, and section 4.2 provides some of the detail of the actual ICEM.

4.1 High level ICEM description

The ICEM is composed of five different types of objects: the packet, the stream, the connection, the host, and the system. Each of these objects are described in detail below. Following the descriptions of each of the objects, we discuss the isolated behavior detection function for connections.

The packet is the only basic object of the system; only packets are observed directly. Once a packet has been pulled off the network, a set of attributes is assigned to the packet. These attributes are the source address, the destination address, the protocol (currently, either TCP/IP or UDP/IP), the source port, the destination port, the number of bytes of data in the packet, the list of bytes for the data, and the time for which the packet was observed. The packet uses its source address, destination address, protocol,

source port, destination port, and time stamp to determine to which stream the packet belongs.

The stream is a complex object consisting of packets. A stream represents a unidirectional flow of data from a process on one computer to a process on another computer. The attributes of a stream are the source address, the destination address, the protocol, the source port, the destination port, the start time, the last update time, the total number of packets, the total number of bytes, and a string matcher. The addresses, protocol, and ports are used to determine which packets belong to each stream. The start time is the time of the first packet observed, and the last update time is the time of the most recent packet for this stream observed. The total number of packets is the number of packets observed for this stream. The total number of bytes is the sum of number of bytes in all the packets for the stream. The string matcher is a complex object used to search the data for specific strings, and it deserves a little more attention.

The string matcher is a list of four-tuples consisting of a string identifier, a counter, a state, and a set of transition functions. The counter indicates the number of times the string, indicated by the string identifier, has been matched in the data from the packets. The state and transition functions describe the deterministic finite automaton (dfa) used to identify the strings. The dfa is constructed using a modified Knuth-Morris-Pratt string matching algorithm. The number of times the strings "Login incorrect" or "Permission denied" have occurred in this data stream can be determined. We have also included a special string in our password files which when observed, indicate that a password file has moved across the network.

A connection is a bidirectional flow of data on the network between a process on one machine and another process on a second machine. It consist of two streams of data. For example, one stream may be the keystrokes an intruder typed in, and the second stream could be the data sent by the computer to the intruder in response to the keystrokes. Our current efforts in intrusion detection has for the most part concentrated on connection objects.

The attributes for a connection are the initiator address, the receiver address, the protocol, an initiator port, a receiver port, a service name, a start time, the last update time, the number of packets from the initiator host, the number of bytes from the initiator host, the strings matched from the initiator host, the number of packets from the receiver host, the number of bytes from the receiver host, and the strings matched from the receiver host. The initiator host is the host which initiated the connection, and the receiver host is the host to which the connection was made.

A host is the network view of a computer. Thus a host in our model is defined by the network traffic generated by it or for it. The host's attributes are the host's internet address, the number of packets sent from the host, the

number of bytes sent from the host, the strings matched in data from the host, the number of packets sent to the host, the number of bytes sent to the host, the strings matched in data sent to the host, total number of connections to or from this host, and the current number of connections to and from this host.

Finally, the system is the summation of all the data on the network. Its attributes are the number of packets observed, the number of bytes observed, the number of each string observed, and the current number of host active on the network.

As mentioned previously, our efforts for detecting intrusive behavior has concentrated on connection objects. For our isolated behavior detection function, we combine the efforts of an anomaly detection system and an expert system.

The anomaly detection system calculates both the probability of the initiator establishing a connection to the receiver host by the given service and normality of the connection. The normality of the connection is based on how closely the given connection compares to connections of the same type of service. We have, therefor, compiled a profile for all of the NSM's currently known services.

The expert system is based on our own personal judgments as to what intrusive behavior should look like. Our personal judgements are, in turn, based on observed past intrusive activity. For example, there are very few legitimate reasons for a password file to cross a network. On the other hand, we have seen password files cross the network in order to crack the passwords on another machine. Thus, seeing a string associated with a password file would cause the connection to be rated very close to intrusive behavior.

The results of the anomaly detection system and the expert system are combined by a weighted average to arrive at a consensus. Currently, we have found the expert system more successful than the anomaly detection system, so the expert system's results are given greater weight.

4.2 Interconnected Computing Environment Model

We now present a portion of the formal ICEM. To present the entire structural grammar for our system would be to lengthy for this particular publication. For the complete ICEM, please contact the author.

The ICEM is defined in part by the structural grammar G = (O,C,S,P), where O, C, S, and, P are defined as follows:

O = {PACKET, STREAM, CONNECTION, HOST,
NETWORK_SYSTEM}
C = {STREAM, CONNECTION, HOST, NETWORK_SYSTEM}
S = NETWORK_SYSTEM

and P is defined to be the following set of production rules:

```
SYSTEM -> {HOST}

  HOST -> {CONNECTION}
      where for all c ∈ {CONNECTION}
          ((HOST.host_addr = c.initiator_addr)
          or (HOST.host_addr = c.receiver_addr))

  CONNECTION -> {STREAM}
      where for all s ∈ {STREAM}
          (CONNECTION.protocol = s.protocol,
          ((CONNECTION.initiator_addr = s.src_addr,
          CONNECTION.receiver_addr = s.dst_addr,
          CONNECTION.initiator_port = s.src_port,
          CONNECTION.receiver_port = s.dst_port)
          or (CONNECTION.initiator_addr = s.dst_addr,
          CONNECTION.receiver_addr = s.src_addr,
          CONNECTION.initiator_port = s.dst_port,
          CONNECTION.receiver_port = s.src_port)))

  STREAM -> {PACKET}
      where for all p ∈ {PACKET}
          (STREAM.src_addr = p.src_addr,
          STREAM.dst_addr = p.dst_addr,
          STREAM.protocol = p.protocol,
          STREAM.src_port = p.src_port,
          STREAM.dst_port = p.dst_port)
```

The semantic attributes, A(), for all the objects are as follows:

A(SYSTEM) = {pkts, bytes, strings_matched, current_host_num}

A(HOST) = {host_addr, pkts_from_host, bytes_from_host, strings_matched_from_host, pkts_to_host, bytes_to_host, strings_matched_to_host, current_connection_num}

A(CONNECTION) = {initiator_addr, receiver_addr, protocol, initiator_port, receiver_port, service, start_time, last_update_time, pkts_from_initiator, bytes_from_initiator, strings_matched_from_initiator, pkts_from_receiver, bytes_from_receiver, strings_matched_from_receiver}

A(STREAM) = {src_addr, dst_addr, protocol, src_port, dst_port, start_time, last_update_time, total_pkts, total_bytes, string_matchers}

A(PACKET) = {src_addr, dst_addr, protocol, src_port, dst_port, num_of_bytes, data, time}

The attributes for a packet are determined by a packet analyzer which is similar to a lexical analyzer - it picks tokens out of the stream of data. The attribute functions for all attributes for all objects would be to numerous to be presented here; however, an overview of the attribute function definitions for one one production, HOST -> {CONNECTION}, are given in figure 7.

5 ANALYZING CONNECTIONS

As this project progressed, we found a strong need for tools which would assist a human to examine objects in the system. For example, if someone logs into a computer from a host never seen before, and if that user generates a number of "Permission denied"s, the NSM will generate a very high warning value of suspicious behavior for that connection. However, security officers needs to know more than just the fact that an intrusion has occurred; the security officer also needs to know exactly how the intrusion occurred (eg. a stolen password), what the intruder saw (that is, what data has been compromised), and what changes the intruder may have made (eg. installing a new account for future entries). In other words, the NSM needs to provide tools not to just detect intrusions, but it needs to provide tools to analyze intrusions as well.

To this end, we have developed two tools, transcript and playback, to analyze CONNECTIONS and their associated data STREAMs.

5.1 Transcript

The transcript tool operates on a CONNECTION and generates two ASCII files, each containing the unidirectional data flow associated with each STREAM. A header is placed at the beginning of each file providing information associated with the connection (the initiator host, the receiver host, the service, the time of the connection, strings matched, etc). The user of the NSM can examine these files with an editor or send them out to a printer.

For telnet connections, one file contains the keystrokes the intruder typed in, and the other file contains the data sent to the intruder's screen. The file containing the data sent to the intruder's screen looks very similar to a file created by the UNIX script command. Thus all commands and command results can be observed.

Unfortunately, if the intruder executes a program, such as a visual editor, which generates screen control characters, these linear transcript files can be difficult to read. To solve this problem, we created a second tool, playback, to properly interpret these control characters.

5.2 Playback

The playback tool writes the data, including all screen control data, from one of the STREAMS associated with a CONNECTION directly to the screen. If playback is run in a screen or window environment which is the same as the intruder's, the screen display will be exactly the same as what the intruder sees. Furthermore, since each packet is time stamped, the timing associated with the intrusion is also reproduced.

This program has been used to observe intruders editing password files with an emacs screen editor. Playback has also been used a number of times to observe

intruders communicating with a fellow hacker with the talk program. Information gained by observing these intrusions with the playback program would be difficult, if not impossible, to obtain through linear data printouts.

6 RESULTS FROM THE NSM

The current version of the NSM runs on a Sun-4 workstation, and it is written in C. The NSM consists of four separate modules: *network_recorder*, *network_analyzer*, *transcript*, and *playback*. The *network_recorder* captures the packets off the network, time stamps each packet, and either directs this data to a file or to the *network_analyzer*. The *network_analyzer* accepts data from the *network_recorder* or from a file created by the *network_recorder*, analyzes the ICEM for intrusive activity, and generates a log of objects and their associated warning value. If the network traffic is recorded in a file, *transcript* accepts a connection identifier (which specifies a particular connection in the log file created by *network_analyzer*) and generates the associated ASCII files. Similarly, *playback* accepts a connection identifier and plays the data from the connection directly to the screen.

Currently we operate the NSM in a batch mode. The *network_recorder* runs twentyfour hours every day of the week; each morning we process the previous twentyfour hours of activity with the *network_analyzer*; and we process the top ten to twenty connections with *transcript* to verify the legitimacy or illegitimacy of the connections. Occasionally, *playback* is used to perform further analysis of a connection.

During a period of two months of testing, over 110,000 connections were analyzed, and the NSM identified correctly over 300 of these connections as being associated with intrusive behavior. These intrusive connections were associated with over 40 different computers, at least four different hardware platforms, and at least six different operating system types (including several flavors of UNIX). The majority of these intrusive connections were associated with attempted break-ins. Successful break-ins occurred on six machines made at three different sites. Other incidents included people acquiring password files from other sites, people running password crackers on password files, ex-employees and ex-students trying to access their closed accounts, people reading other people's mail files, people attempting to look around other people's directories, and people using other people's accounts.

Out of all these recorded connections of intrusive activity, only eight of them were detected by system administrators or the user community. Of the eight identifications, five of them were made by members of our security laboratory who observed attempted penetration of our security lab machines. Therefore, only three of the roughly 300 connections associated with intrusive behavior were identified without the aid of the NSM.

7 FUTURE RESEARCH

The work on the NSM is far from over. The higher order objects of the ICEM, and their associated secondary functions to detect intrusive behavior will probably be modified as we gain more experience in intrusion detection. Secondary functions to detect other behaviors are being considered as well. These functions include those which identify specific classes of intrusive behavior, identify objects behaving within a specified policy, and identify failures in network components.

Beyond the work for a single NSM, we plan to look at the coordination of the efforts of multiple NSMs distributed over different parts of a wide area network. Coordinating effort would allow successful tracking of intruders as they cross network boundaries. Furthermore, the combination of multiple monitors would provide another method of aggregating warnings of subtle attacks distributed over disparate areas.

Our current work includes the integration of the NSM into a distributed intrusion detection system. This system, called DIDS (Distributed Intrusion Detection System), combines the efforts of host based and network based monitors to benefit from the advantages of both types of monitors.

Finally, we are looking at casting other environments into the system model. For example, the dynamic environment of the host can be modeled with the following objects: operating system calls, processes, users, and the host. Behavior functions to determined whether each operating system call, process, user, and host is behaving in an intrusive manner could draw upon the work by SRI, LANL, TRW, and others designing host based intrusion detection systems. The process described here could also be used to detect behaviors in dynamic and complex mechanical and biological systems.

8 CONCLUSIONS

As government reports, recent books, and the popular media have stated, our computers systems are vulnerable to attack. Authentication and access control mechanisms to prevent intrusive activity have obviously not been wholly successful, so a second layer of defense, intrusion detection, is needed. We took on the task of developing an intrusion detection system capable of simultaneously detecting intrusive behavior on many hosts running many versions of operating systems. To solve this problem, we developed a model to detect behaviors in dynamic and complex systems. We then mapped the interconnected computing environment into this model, and we defined secondary functions to detect intrusive behavior on objects in our model. Finally we have designed tools to allow a user to examine in detail connections on the network. Current results show this system to be very successful.

References

1. G.V. Bochmann, "Semantic Evaluation from Left to Right," Communications of the ACM, vol. 19, no. 2, pp. 55-62, Feb. 1976.

2. D.E. Denning, "An Intrusion Detection Model," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 2, pp. 222-232, Feb. 1987.

3. D.E. Denning, a conversation with L. Todd Heberlein, December 6, 1990

4. Department of Defense Trusted Computer System Evaluation Criteria, Dept. of Defense, National Computer Security Center, DOD 5200.28-STD, Dec. 1985.

5. C. Dowell and P. Ramstedt, "The COMPUTERWATCH Data Reduction Tool," *Proc. 13th National Computer Security Conference*, pp. 99-108, Washington, D.C., Oct 1990.

6. D.B. Guralnik, ed. Webster's New World Dictionary, (Simon and Schuster, 1980).

7. L.T. Heberlein, et al., "A Network Security Monitor," *Proc. 1990 Symposium on Research in Security and Privacy*, pp. 296-304, May 1990.

8. D.E. Knuth, "Semantics of Context-Free Languages," *Math Systems Th. 2* (1968), 127-145. Correction appears in *Math Systems Th.5* (1971),95.

9. T.F. Lunt, et al., "A Real Time Intrusion Detection Expert System (IDES)," Interim Progress Report, Project 6784, SRI International, May 1990.

10. M.M. Sebring, et al., "Expert Systems in Intrusion Detection: A Case Study," *Proc. 11th National Computer Security Conference*, pp. 74-81, Oct. 1988.

11. S.E. Smaha, "Haystack: An Intrusion Detection System," *Proc. IEEE Fourth Aerospace Computer Security Applications Conference*, Orlando, FL, Dec. 1988.

12. W.T. Tener, "Discovery: an expert system in the commercial data security environment," *Security and Protection in Informations Systems: Proc. Fourth IFIO TC11 International Conference on Computer Security*, North-Holland, May 1988.

13. H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," *Proc, Symposium on Research in Security and Privacy,* pp. 280-289, Oakland, CA, May 1989.

14. J.R. Winkler, "A Unix Prototype for Intrusion and Anomaly detection in Secure Networks," *Proc. 13th National Computer Security Conference*, pp. 115-124, Washington, D.C., Oct. 1990.

$N \rightarrow L.L$
$N \rightarrow L$
$L \rightarrow LB$
$L \rightarrow B$
$B \rightarrow 1$
$B \rightarrow 0$

Figure 1

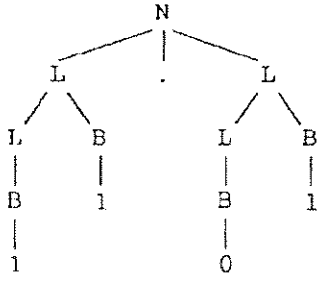


Figure 2

$N \rightarrow L_1.L_2$	$v(N) = v(L_1) + V(L_2)/2^{l(L_2)}$
$N \rightarrow L$	$v(N) = v(L)$
$L_1 \rightarrow L_2B$	$v(L_1) = 2v(L_2) + v(B)$, $l(L_1) = l(L_2)+1$
$L \rightarrow B$	$v(L) = v(B)$, $l(L) = 1$
$B \rightarrow 1$	$v(B) = 1$
$B \rightarrow 0$	$v(B) = 0$

Figure 3

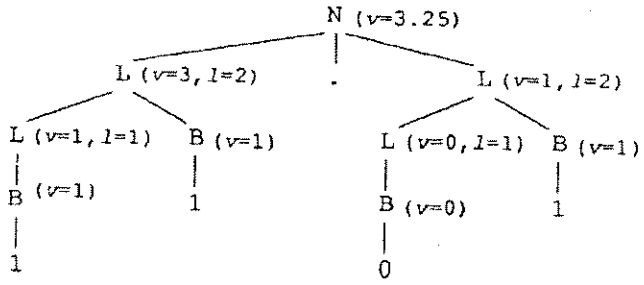


Figure 4

$$\frac{W_1*(\text{isolated result}) + W_2*(\text{super-component results}) + W_3*(\text{sub-component results})}{W_1 + W_2 + W_3}$$

Figure 5

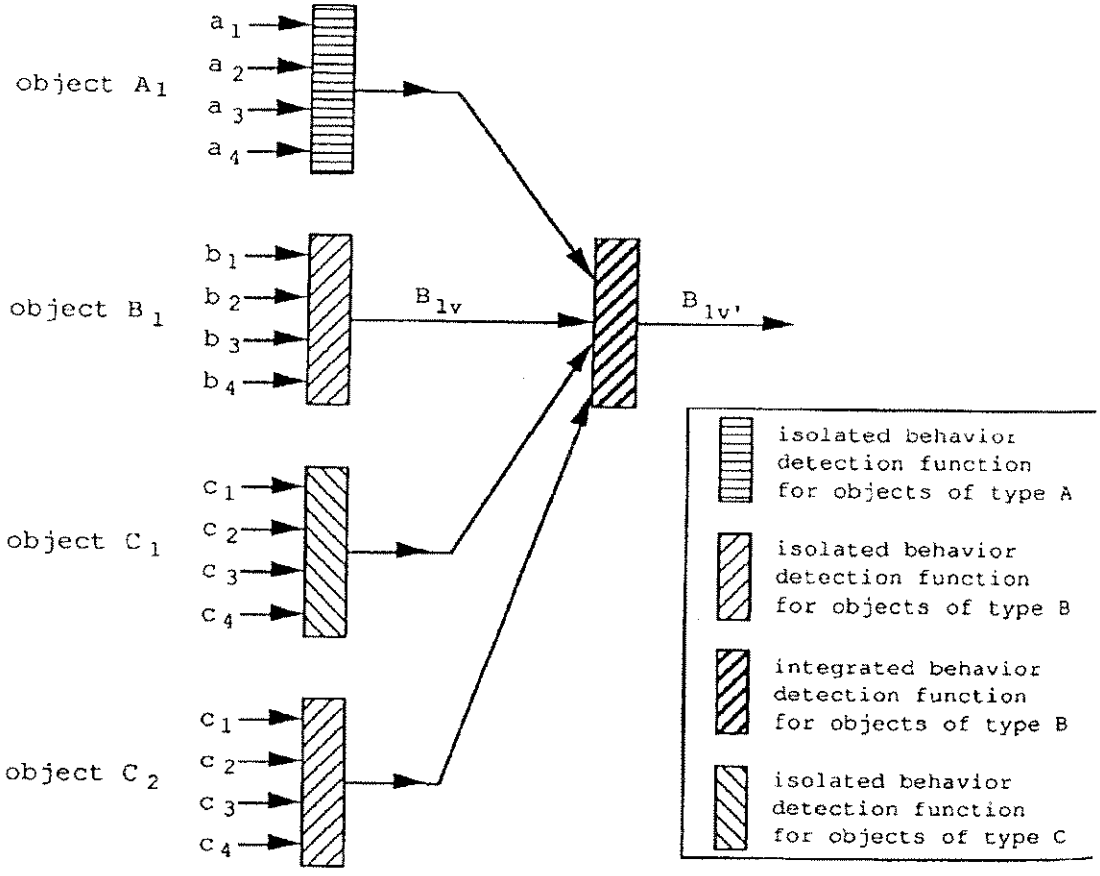


Figure 6

$$HOST.pkts\_from\_host = \sum_{c \in C1} c.pkts\_from\_initiator$$

$$HOST.bytes\_from\_host = \sum_{c \in C1} c.bytes\_from\_initiator$$

$$HOST.strings\_matched\_from\_host = \sum_{c \in C1} c.strings\_matched\_from\_initiator$$

$$HOST.pkts\_to\_host = \sum_{c \in C2} c.pkts\_from\_initiator$$

$$HOST.bytes\_to\_host = \sum_{c \in C2} c.bytes\_from\_initiator$$

$$HOST.strings\_matched\_to\_host = \sum_{c \in C2} c.strings\_matched\_from\_initiator$$

where for all $c \in C1$

$(c \in \{CONNECTION\},$
$HOST.host\_addr = c.initiator\_addr)$

and for all $c \in C2$

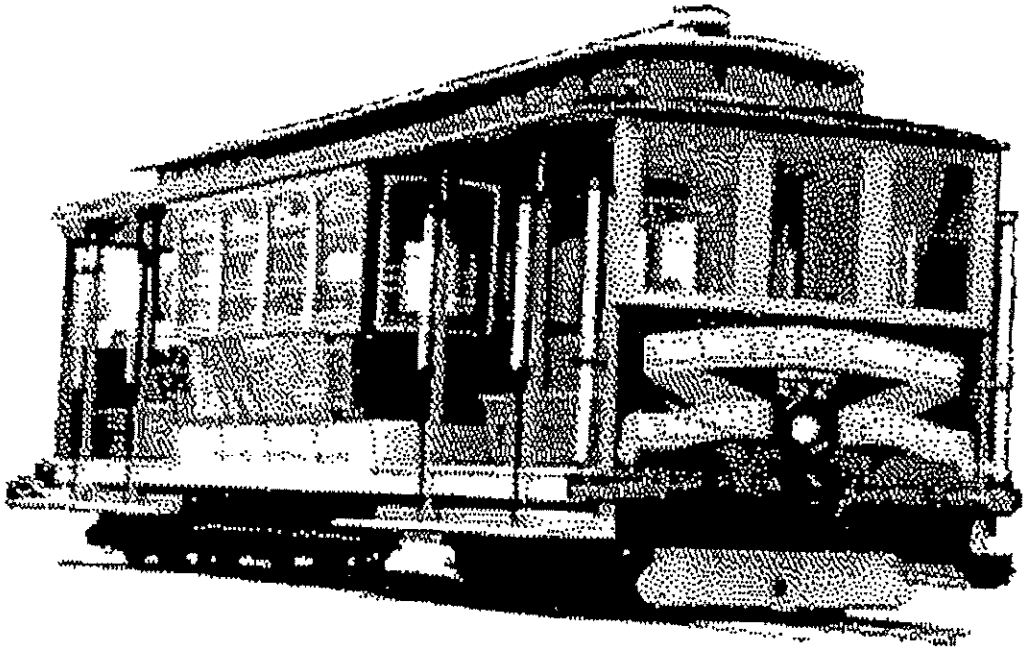
$(c \in \{CONNECTION\},$
$HOST.host\_addr = c.receiver\_addr)$

Figure 7

CONF-9105126

14th Department of Energy Computer Security Group Conference

Proceedings



Concord, California *May 7 - 9, 1991*

Lawrence Livermore National Laboratory

"Computer Security - A Personal Commitment"

U.S. Department of Energy
Office of Administration and Human Resource Management
Office of Information Resources Management Policy, Plans and Oversight
and
Office of Security Affairs
Office of Safeguards and Security